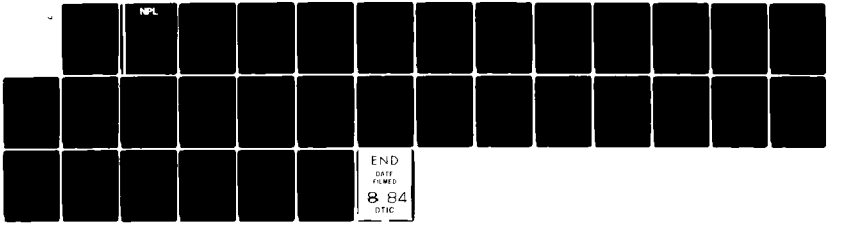
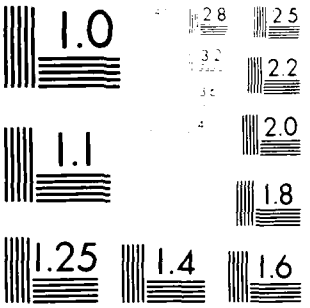
AD-A142 655 AN INTRODUCTION AND HANDBOOK FOR STANDARD SYNTACTIC METALANGUAGE(U) NATIONAL PHYSICAL LAB TEDDINGTON (ENGLAND) DIV OF INFORMATION TECHNOLOGY AND COMPUTING 1/1

UNCLASSIFIED R S SCOWEN FEB 83 NPL/DITC-19/83 F/G 5/7 NL





(2)

NPL Report DITC [illegible]/83
February 1983

# An introduction and handbook for the standard syntactic metalanguage

by
R S Scowen

ISSN 0262-5369

National Physical Laboratory
Teddington, Middlesex TW11 OLW, UK



Approved on behalf of Director, NPL, by Mr E L Albasiny
Superintendent, Division of Information Technology and Computing

NPL Report DITC 17/83

February 1983

NATIONAL PHYSICAL LABORATORY

# An introduction and handbook for the standard syntactic metalanguage

by

R S Scowen

Division of Information Technology and Computing

Abstract

British Standard BS 6154 defines a standard syntactic metalanguage. This report first explains what a metalanguage is and why it is useful; the rest of the report is a guide and summary of the standard metalanguage. Many examples are taken from subjects outside computing, others come from Fortran or Pascal, two of the commonest programming languages.

Important features of the standard metalanguage are that it can be typed easily and processed by computers.

This report is based on an earlier one, NPL Report DNACS 47/81, now out of print.

CONTENTS

# AN INTRODUCTION TO THE STANDARD SYNTACTIC METALANGUAGE

Languages enable communication, sometimes between people, sometimes animals, sometimes computers. The syntax of a language defines precisely those sequences of words or symbols which are grammatically valid. If you want to describe or define a language, it is natural to use English which is easy to write and understand. However there are disadvantages, for example it is easy to be ambiguous without realizing it. The subsequent confusion or misunderstanding will increase costs and waste time; it may even result in disaster.

A better strategy is to define your language using a notation specially designed for describing the structure or meaning of languages. Any such notation is called a metalanguage, thus a syntactic metalanguage is a notation for specifying a syntax (or format) precisely. It should be a valuable weapon in the armoury of every computer scientist but although the concepts are well known to compiler writers and language designers, many different notations are used with the result that some programmers are unfamiliar with the ideas.

The syntax definition of a language serves three different purposes:

- it names the various parts of the language,
- it shows how to construct sentences of the language that are syntactically valid, and
- it also indicates the syntactic structure of any given sentence of the language.

Note that a syntactic metalanguage is useful not only for the syntax of a programming or command language, but whenever a formally defined syntax is required, e.g. the format for references in scientific reports, or the interface between two programs.

A British Standard (BS6154) defines a standard syntactic metalanguage. One important feature is that it can be typed on most computer terminals and processed by computers.

The existence of a British Standard metalanguage makes life much simpler for project managers and data processing managers when they must specify documentation standards. Previously, if they specified "Use Backus-Naur-Form", there would be cries of anguish, "It's primitive", "It's long-winded", or, "My notation is much clearer and shorter". The result was likely to be argument and vacillation. One team defining a candidate language for the US Department of Defense spent six months deciding on a syntactic metalanguage.

Another company has adopted BS 6154 with great success, "Previously people thought they knew the requirements, now they are certain what is wanted".

This report is a beginner's guide to the standard metalanguage. Often the simplest way of understanding a new idea is by studying its application to something that is already familiar. So most of the examples define elements of Fortran or Pascal, two of the commonest programming languages. Each syntax rule names part of the language (called a non-terminal symbol of the language) and then defines all its possible forms.

## THE BASIC FORM OF A SYNTAX RULE

Each syntax rule starts with the name (known formally as a metaidentifier and pronounced meta-identifier) of the structure being defined. Then comes = (equals), the definition follows, and the syntax rule ends with ; (semicolon).

A metaidentifier is one or more words formed with letters and digits, with the first character being a letter. Sometimes the definition consists, wholly or partly, of explicit characters which always appear in the structure being defined, a group of such characters is called a terminal symbol. They are represented by themselves and enclosed by " (quotation mark). For example:

```
continue statement = "CONTINUE" ;
```

states that a continue statement is formed by the eight characters:

```
CONTINUE
```

Note that a terminal symbol of a language is like an atom that cannot be split into smaller components of the language.

The metalanguage refers to the characters of a terminal symbol together with the enclosing apostrophes as a terminal string.

### Alternative definitions

When there are several possible definitions, the alternatives are listed one after the other and separated by | (vertical line). For example:

```
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
```

states that a digit is any one of the ten characters

```
0 1 2 3 4 5 6 7 8 9
```

### Using symbols that have already been defined

When a metaidentifier occurs after the = (equals) of a syntax rule, it represents any sequence of symbols defined by a syntax rule that starts with that metaidentifier. For example:

```
octal digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" ;
digit = octal digit | "8" | "9" ;
```

states that an octal digit is any one of the eight characters

```
0 1 2 3 4 5 6 7
```

and (indirectly) that a digit is any one of the ten characters

```
0 1 2 3 4 5 6 7 8 9
```

A sequence of items

A definition may consist of several terminal strings and metaidentifiers in a particular order. In this case the symbols are listed in the correct order and separated from each other by a , (comma). For example:

```
assignment statement = variable, "=", expression ;
```

states that an assignment statement is a variable followed by = (equals) followed by an expression. This rule says nothing about the form of a variable or an expression; other rules to define them will appear in a complete syntax.

A specific number of items

In Fortran a label at the start of a statement always contains five characters with each character a digit or blank character, this could be defined by the rules:

```
blank = " " ;
label char = digit | blank ;
label field = label char, label char, label char,
    label char, label char ;
```

A simpler way specifies the exact number of label chars by a preceding integer and * (asterisk). Thus the third syntax rule can be rewritten:

```
label field = 5 * label char ;
```

Brackets group items together

In mathematics ( and ) (brackets) are used to group items together. Brackets have a similar meaning in the metalanguage, for example the previous three rules could be written:

```
label = 5 * ( digit | " " ) ;
```

Precedence

So far the meaning of several symbols in the metalanguage has been explained without saying whether a syntax rule can contain every sort of symbol. For example, would

```
label = 5 * digit | " " ;
```

be a valid syntax rule? If so, what does it signify? In fact it is a valid rule, but means the same as the bracketed rule

```
label = ( 5 * digit ) | " " ;
```

This is because the British Standard defines * to have a higher precedence in the metalanguage than |. This idea of precedence occurs also in Fortran and Pascal where * and / have a higher precedence than + and -. Page 7 contains a complete precedence table for the metalanguage.

All brackets override the normal precedence. Used wisely, they make languages easier to understand by reducing the number of syntax rules

and shortening the language definition. But too many brackets make syntax rules complicated and obscure.

An optional item

Many constructions in Fortran have symbols that are optional, for example the increment in a do-statement need not be specified when its value is one. Optional symbols are specified by enclosing them in [ and ] (square brackets), e.g:

```
do statement = "DO", label, variable, "=", initial value, ",",
    final value, [",", increment] ;
```

An indefinite number of items

Sometimes there is no logical limit to the length of a structure, for example a Fortran arithmetic expression can be as long as you like (although strictly there is a limit because a statement must not have more than 19 continuation lines). Symbols that are optional or that can be repeated any number of times are enclosed in { and } (curly brackets), e.g:

```
arithmetic expression = [sign], arithmetic primary,
    {arithmetic operator, arithmetic primary} ;
```

states that an arithmetic expression is one or more arithmetic primaries, any two arithmetic primaries are separated by an arithmetic operator, and there may be a sign at the start of the arithmetic expression.

Everything except an exceptional case

Sometimes a definition would be quite simple except for a few special cases. The metalanguage permits such definitions to be expressed by giving the general rule, then - (minus sign), then the exceptional cases, for example:

```
Fortran 77 continuation line = 5 * " ",
    (character - (" " | "0")), 66 * [character] ;
```

This rule states that a Fortran 77 continuation line starts with 5 blanks, the sixth character must not be a blank or zero, and there must not be more than 72 (= 5 + 1 + 66) characters altogether.

In Fortran 66, the definition of a continuation line is more complicated, i.e:

```
Fortran 66 continuation line = (character - "C"),
    4 * character, (character - (" " | "0")),
    66 * [character] ;
```

This rule states that a continuation line must not start with C, there must be at least 6 characters, the sixth character must not be a blank or zero, and there must not be more than 72 ( = 1 + 4 + 1 + 66) characters altogether.

## Syntax processors and the need for comments

The metalanguage has been designed so that computer programs can be used to process the syntax of a language. It is often beneficial if explanatory remarks can be added to the syntax without affecting the results of a syntax processor. It is also natural to mention the extra semantic restrictions that cannot easily be expressed in a syntax. For reasons such as this the syntactic metalanguage includes a comment notation. Any text in a comment has no formal effect on the language defined by a syntax. A comment starts with (* (left parenthesis, asterisk) and ends with *) (asterisk, right parenthesis), e.g.

```
integer = digit, {digit}
    (* The maximum magnitude of an integer in IBM Fortran IV
    on 360/370 computers is 2147483647 (i.e. 2^31 - 1) *) ;
```

However (* or *) in a terminal string stand for themselves and do not start or end a comment.

## One or more repetitions

A shorter way of defining an integer as one or more digits is:

```
integer = {digit} - empty ;

empty = ;
```

Or even more concisely:

```
integer = {digit}- ;
```

This definition is equivalent because the general case is zero-or-more digits and the exceptional invalid case is empty.

## Special sequences

It is always difficult to foresee all possible uses for a new notation, so the syntactic metalanguage includes a method for extending the notation. In any syntax rule the meaning of any text that starts and ends with ? (question mark) is not defined by the standard. Such text is called a special sequence; when this notation is used there should be an explanation with the syntax of the language telling you how to interpret it.

## More on terminal strings

A terminal symbol can be enclosed by ' (apostrophe) instead of " (quotation-mark). This allows us to define terminal symbols containing a quotation-mark.

Note that there must be at least one character between the apostrophes or quotation marks of a terminal string.

For example, the definition of a terminal symbol in the syntactic metalanguage is:

```
terminal string = '"', { character - '"' }-, '"'
    | "'", { character - "'" }-, "'" ;
```

## READING SYNTAX RULES

It is not difficult to translate and read a syntax rule in a rather strange form of English. Table A gives an equivalent English phrase for each character in the metalanguage. Unfortunately we cannot forget about the metalanguage and define our languages using only the equivalent English phrases because the English version becomes ambiguous with complicated syntax rules. And remember, avoiding ambiguity is one of the reasons for having a formal metalanguage.

TABLE A -- READING A SYNTAX RULE

| METALANGUAGE SYMBOL | ENGLISH PHRASE |
|---|---|
| metaidentifier | A or an ... |
| terminal | The character(s) ... |
| ; | . (The end of a rule) |
| = | Is |
| \| | Or |
| , | Followed by |
| - | Except |
| * | Occurrences of |
| { } | Any number of |
| [ ] | Optional |

Examples

The examples are taken from the metalanguage's own definition. Each syntax rule is followed by an English 'translation'. The words corresponding to the metalanguage symbols have been written in upper case so that it is easier to compare the original rule with its translation. Sometimes the order of words has been altered slightly so that the translated rule reads better, sometimes, as stated earlier, the translation is ambiguous.

```
syntax  =  syntax rule, {syntax rule} ;
A syntax IS A syntax rule FOLLOWED BY ANY NUMBER OF A syntax rule.

factor  =  [integer, "*"], primary ;
A factor IS AN OPTIONAL integer FOLLOWED BY * FOLLOWED BY A primary.

optional sequence  =  "[", definitions list, "]" ;
AN optional sequence IS [ FOLLOWED BY A definitions list FOLLOWED BY ].

special sequence  =  "?", {character - "?"}, "?" ;
A special sequence IS ? FOLLOWED BY ANY NUMBER OF A character EXCEPT ?
    FOLLOWED BY ?.
```

## THE PRECEDENCE OF THE METALANGUAGE OPERATORS

Table B gives the precedence of the various metalanguage operators; the higher the symbol in the table, the higher its precedence.

All brackets override the normal precedence.

TABLE B -- PRECEDENCE OF METALANGUAGE OPERATORS

| METALANGUAGE SYMBOL | MEANING |
|---|---|
| * | Repeat |
| - | Except |
| , | Concatenate |
| \| | Or |
| = | Defines |
| ; | End of rule |

Example

The Fortran 66 continuation line (already defined on page 4) could have been defined with fewer brackets.

```
Fortran 66 continuation line = character - "C",
    4 * character, character - (" " | "0"),
    66 * [character] ;
```

## LIMITATIONS OF THE SYNTACTIC METALANGUAGE

The main restriction of the standard metalanguage is that the language being defined must be linear, i.e. the symbols in a sentence of the language can be placed in a line reading from one end to the other. For example knitting patterns and recipes in cooking are linear languages but electric circuit diagrams are not.

A further limitation is that the notation is inadequate for defining complex grammars that also define semantic restrictions on the possible sequences of symbols. However a way has been left open for the metalanguage to be suitably extended; further details are beyond the scope of this beginner's guide.

Naturally, the metalanguage, like most other notations, can be misused. For example it does not prevent someone from trying to define an unparsable or ambiguous language.

## THE CHARACTER SETS

Syntax rules in the standard metalanguage are written using the standard ISO 7-bit character set. This is similar to the ASCII character set and is available on many computer systems. Syntax rules can also be printed on conventional office typewriters because alternative characters are defined for those symbols that are unavailable on some typewriters. Table C gives all the possible representations for each metalanguage symbol.

TABLE C -- THE METALANGUAGE CHARACTER SETS

| METALANGUAGE SYMBOL | POSSIBLE REPRESENTATIONS | |
|---|---|---|
| concatenate symbol | , | |
| defining symbol | = | |
| definition separator symbol | ; | / ! |
| end comment symbol | *) | |
| end group symbol | ) | |
| end option symbol | ] | /) |
| end repeat symbol | } | :) |
| except symbol | - | |
| first quote symbol | ' | |
| repetition symbol | * | |
| second quote symbol | " | |
| special sequence symbol | ? | |
| start comment symbol | (* | |
| start group symbol | ( | |
| start option symbol | [ | (/ |
| start repeat symbol | { | (: |
| terminator symbol | ; | . |

The characters required for the metalanguage are:

letters digits = , - * ( ) ?
; or .
| or / or !
/ or both [ ]
: or both { }
' or " (Both characters are desirable)

# EXAMPLES

## 1. A few one-line definitions

```
space = " "
  (* defines a non-terminal symbol called 'space' to represent
          a space character *) ;

visible character = character - space
  (* defines a visible character to be
          any character except a space *) ;

digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  (* defines a digit to be any of the ten characters
          0 1 2 3 4 5 6 7 8 9 *) ;

integer = {digit}-
  (* defines an integer to be one or more digits *) ;

vowel = "A" | "E" | "I" | "O" | "U"
      (* defines a vowel to be one of the five characters
          A E I O U *) ;
```

## 2. Telephone numbers

```
london telephone number = "01-", exchange, " ", number ;

  exchange = 3 * digit ;

  number = 4 * digit ;
```

## 3. The Forsyth notation for recording a chess position

The Forsyth notation provides a simple method of recording any chess position. Although a position recorded in this way is not immediately obvious to a reader, it is much more quickly written than a diagram with pictures of pieces.

The Forsyth notation describes a chess position by specifying the pieces on each row. It starts at the top of a diagram, i.e. Black's first rank, and proceeds row by row ending at Black's eighth rank. Each row is described from left to right, i.e from the Queen's Rook file to the King's Rook file. A White piece is written as an upper case letter, and a Black piece a lower case letter. The abbreviations are:

K = King, Q = Queen, R = Rook, B = Bishop, N = Knight, P = Pawn.

A number indicate a sequence of empty squares or rows.

SYNTAX

```
Forsyth chess position  =  row,  { "/",  row },  ;

  row  =  one or more empty rows  |  single row ;

    one or more empty rows
      =  "8"  |  "16"  |  "24"  |  "32"  |  "40"  |  "48"  |  "56" ;

    single row  =  [ one or more empty squares ],
      { chess piece,  [ one or more empty squares ] }- ;

      one or more empty squares
        =  "1"  |  "2"  |  "3"  |  "4"  |  "5"  |  "6"  |  "7" ;

      chess piece  =  black piece  |  white piece ;

      black piece  =  "k"  |  "q"  |  "r"  |  "b"  |  "n"  |  "p"
        (*  In manuscripts, each letter representing a black piece is
            often circled rather than being written in lower case *) ;

      white piece  =  "K"  |  "Q"  |  "R"  |  "B"  |  "N"  |  "P" ;
```

EXAMPLES

(1) The position at the start of a game of chess is:
rnbqkbnr / pppppppp / 32 / PPPPPPPP / RNBQKBNR

(2) The position after the moves:
1 P-K4  P-K4  2 B-B4
rnbqkbnr / pppp1ppp / 8 / 4p3 / 2B1P3 / 8 / PPPP1PPP / RNBQK1NR

(3) The position after the moves:
1 P-K4  P-K4  2 B-B4  B-B4  3 Q-R5  N-QB3??  4 Q-R5 mate
r1bqk1nr / pppp1Qpp / 2n5 / 2b1p3 / 2B1P3 / 8 / PPPP1PPP / RNB1K1NR

## 4. Bibliographic references

BS 6154 refers to three scientific papers that strongly influenced the standard.

This example shows how the metalanguage can specify the form of references and thus help editors maintain a consistent style.

The references as printed in BS 6154 are:

> NAUR, P.(Ed). Revised Report on the Algorithmic Language ALGOL 60. Computer Journal, January, 1963, Vol 5, No. 4 pp 349-367.
>
> WIRTH, N. What can we do about the unnecessary diversity of notation for syntactic definitions? Comm ACM, November 1977, Vol 20, No 11, pp 822-823.
>
> VAN WIJNGAARDEN, A., MAILLOUX, B.J., PECK, J.E.L., KOSTER, C.H.A., SINTZOFF, M., LINDSEY, C.H., MEERTENS, L.G.L.T. and FISKER, R.G. Revised Report on the Algorithmic Language ALGOL 68, Acta Informatica, 1976, Vol 5, parts 1-3, (also published in SIGPLAN Notices, May 1977, Vol. 12, No. 5, pp 1-70).

```
SYNTAX

reference = authors, title, sources, point ;

  authors
      = [ author, { comma, author }, space, "and", space ],
      author, [ "(Ed)." ] ;

    author = surname, comma, { initial }- ;
    surname = upper case letter, { [ space ], upper case letter } ;
    initial = upper case letter, point ;

  title = one or more words, { comma | "?", space } ;

  sources
      = source, [ comma, "(also published in ", source, ")" ] ;

    source = journal, date, volume, [ parts ], [ pages ] ;

      journal = one or more italic words, comma ;
      date = [ month, space ], year ;
      volume = "Vol", [ point ], space, volume number, comma ;
      parts = "No", [ point ], space, integer, { comma | space }
            | "parts", space, first part, hyphen, last part ;
      pages = "pp", space, first page, hyphen, last page ;

        month = upper case letter, { lower case letter }- ;
        year = 4 * digit, comma ;
        volume number = bold face integer ;
        first part = integer ;
        last part = integer ;
        first page = integer ;
        last page = integer ;

(* Common definitions *)

comma = ",", space ;
hyphen = "-" ;
```

```
point     =  "." ;
space     =  " "  |  new line ;

(*  Definitions not given here  *)

    bold face integer
    digit
    integer
    lower case letter
    one or more italic words
    one or more words
    upper case letter
```

## 5. PS 6154 - syntactic metalanguage

This example defines the syntax of the standard syntactic metalanguage using itself. Many of the syntax rules include a comment to explain their meaning; inside a comment a metaidentifier is enclosed in angle brackets < and > to avoid confusion with similar English words. The non-terminal symbols <letter>, <decimal digit> and <character> are not defined. The position of <comments> is stated in a comment but not formally defined.

*Characters such as spaces and new lines affect the language defined by a syntax only when they appear in a terminal string.*

```
syntax = syntax rule, {syntax rule} ;
syntax rule = metaidentifier, "=", definitions list, ";"
    (* A <syntax rule> defines the sequences of symbols represented by
       a <metaidentifier> *) ;
definitions list = single definition, {"|", single definition}
    (* | separates alternative <single definitions> *) ;
single definition = term, {",", term}
    (* , separates successive <terms> *) ;
term = factor, ["-", exception]
    (* A <term> represents any sequence of symbols that is defined
       by the <factor> but not defined by the <exception> *) ;
exception = factor
    (* A <factor> may be used as an <exception> if it could be
       replaced by a <factor> containing no <metaidentifiers> *) ;
factor = [integer, "*"], primary
    (* The <integer> specifies the number of repetitions of
       the <primary> *) ;
primary = optional sequence | repeated sequence | special sequence
    | grouped sequence | metaidentifier | terminal string ;
optional sequence = "[", definitions list, "]"
    (* The brackets [ and ] enclose symbols which are optional *) ;
repeated sequence = "{", definitions list, "}"
    (* The brackets { and } enclose symbols which may be
       repeated any number of times *) ;
grouped sequence = "(", definitions list, ")"
    (* The brackets ( and ) allow any <definitions list>
       to be a <primary> *) ;
terminal string = '"', character - '"', {character - '"'}, '"'
    | "'", character - "'", {character - "'"}, "'"
    (* A <terminal string> represents the <characters> between the
       quote symbols " " or ' ' *) ;
metaidentifier = letter, {letter | decimal digit}
    (* A <metaidentifier> is the name of a syntactic element of the
       language being defined *) ;
integer = decimal digit, {decimal digit} ;
special sequence = "?", {character - "?"}, "?"
    (* The meaning of a <special sequence> is not defined in the
       standard metalanguage. *) ;
comment = "(*", {comment symbol}, "*)"
    (* A comment is allowed anywhere outside a <terminal string>,
       <metaidentifier>, <integer> or <special sequence> *) ;
comment symbol
    = comment | terminal string | special sequence | character ;
```

## 6. The syntactic metalanguage in an alternative representation

```
SYNTAX = SYNTAX RULE, (: SYNTAX RULE :) .
SYNTAX RULE = METAIDENTIFIER, '=', DEFINITIONS LIST, '.' .
DEFINITIONS LIST = SINGLE DEFINITION, (: '/', SINGLE DEFINITION :) .
SINGLE DEFINITION = TERM, (: ',', TERM :) .
TERM = FACTOR, (/ '-', EXCEPTION /) .
EXCEPTION = FACTOR .
FACTOR = (/ INTEGER, '*' /), PRIMARY .
PRIMARY = OPTIONAL SEQUENCE / REPEATED SEQUENCE / SPECIAL SEQUENCE
    / GROUPED SEQUENCE / METAIDENTIFIER / TERMINAL / .
OPTIONAL SEQUENCE = '(/', DEFINITIONS LIST, '/)' .
REPEATED SEQUENCE = '(:', DEFINITIONS LIST, ':)' .
GROUPED SEQUENCE = '(', DEFINITIONS LIST, ')' .
TERMINAL
     = "'", CHARACTER - "'", (: CHARACTER - "'" :), "'"
    / '"', CHARACTER - '"', (: CHARACTER - '"' :), '"' .
METAIDENTIFIER = LETTER, (: LETTER / DIGIT :) .
INTEGER = DIGIT, (: DIGIT :) .
SPECIAL SEQUENCE = '?', (: CHARACTER - '?' :), '?' .
COMMENT = '(*', (: COMMENT SYMBOL :), '*)' .
COMMENT SYMBOL = COMMENT / TERMINAL / SPECIAL SEQUENCE / CHARACTER .
```

## WRITING SYNTAX RULES CLEARLY

The syntactic metalanguage is only a notation. It does not prevent you from defining pathological languages or from writing opaque clumsy definitions. But whatever the language being defined, you can help your readers to understand it by setting out the rules as clearly as possible.

### Keep rules simple

Do not write rules that are too complicated. If necessary split a complex rule into several simpler rules.

### A clear layout

Spaces, new lines and other non-printing characters outside a terminal string have no effect on a syntax, so insert them freely to make the meaning evident.

(1) Start each syntax rule on a new line.

(2) When a syntax rule is too long to fit on a single line, start all lines after the first with extra spaces.

These first two rules ensure that it is easy to see the extent of each rule.

(3) Put spaces before and after each metasymbol as follows:

| BEFORE | METASYMBOLS | AFTER |
|---|---|---|
| | ; | New line |
| Space | = \| - * | Space |
| | , | Space |

(4) Whenever possible, never start a new line in the middle of a metaidentifier. When a definition will not fit on one line, break the definition at a space before a metasymbol which has least depth of brackets. The essential difference between the following two Pascal rules is then obvious:

```
block
     =  label declaration part,
    constant definition part,
    type definition part,
    variable declaration part,
    procedure and function declaration part,
    statement part ;
formal parameter section
     =  value parameter specification
    | variable parameter specification
    | procedural parameter specification
    | functional parameter specification ;
```

Use meaningful metaidentifiers

Metaidentifiers are names given to components of the language, and are completely arbitrary, for example:

```
b   =  ldp, cdp, tdp, vdp, pafdp, sp ;
fps  =  vlps | vrps | prps | fnps ;
```

could replace the two rules for 'block' and 'formal-parameter-specification' given above. With abbreviations like this the whole of Pascal could be defined in a single page, but only with almost complete loss of clarity.

Place comments sensibly

Put any comments about a rule before the final semicolon so that even a computer knows which rule the comment is referring to.

Order rules sensibly

The rules for a language can be given in any order, but a language like Pascal or Fortran has more than a hundred rules, and a reader is helped if the rules are in some sort of logical order. For example:

(1) Group together the rules for each particular part of the language.
(2) Follow each metaidentifier with a comment indicating whereabouts in the language definition that part of the language is defined.
(3) Put all the rules in alphabetical order. A program can do this.

Provide a cross-reference index

A cross-reference is another helpful aid in understanding large languages; it shows where each metaidentifier and terminal symbol in the language is used in other syntax rules.

A cross-reference index can also be made by a program.

Explain complex languages with syntax diagrams

The standard for Fortran 77 explains the syntax of the language with syntax diagrams, i.e. each non-terminal symbol is defined by a diagram looking like a network of railway lines. Any smooth path from start to finish indicates a valid sequence of symbols forming the non-terminal symbol. This notation is easy to understand but more difficult to write and type.

Syntax diagrams are probably most easily prepared by a computer program that reads syntax rules as data.

Clearly distinguish metaidentifiers in explanatory text

Metaidentifiers are usually one or more English words. The explanation of a language may be confusing unless the metaidentifiers are clearly distinguishable from ordinary English words. Several notations are possible:

(1) Surround metaidentifiers by < and > (angle brackets);

(2) Write a hyphen instead of a space between the separate words of a metaidentifier;
(3) Use a distinctive type font for metaidentifiers;
(4) Underline metaidentifiers.

Take care to define whatever notation you adopt; the standard itself is silent on the matter.

Logically defined rules

It is sensible if the language definition reflects the structure of the language, only then can a reader understand it easily.

## APPENDIX A - SOME FURTHER DETAILS

This introduction to the metalanguage has simplified the description by omitting some details. This appendix describes features which might have been confusing on a first reading.

### Multiple definitions of metaidentifiers

Syntax rules starting with the same metaidentifier provide another method of making alternative definitions. This is convenient when a language contains several levels. For example, Fortran 77 has two levels. At the subset level

```
logical operator = ".NOT." | ".AND." | "OR" ;
```

The full language has two additional logical operators:

```
logical operator = ".EQV." | ".NEQV." ;
```

### Lexical conventions

A language defined by the metalanguage may have lexical rules completely different from the metalanguage itself. For example in the syntactic metalanguage a metaidentifier may contain spaces or other gaps to improve its readability, and a line may be of any length and start with any number of spaces. However, the metalanguage may be used to define Fortran statements where each line has 72 characters and a label occupies the first 5 characters of a line; or to define Pascal identifiers where no gaps are allowed in an identifier or integer.

### A restriction on exceptions

As stated earlier a metaidentifier can be defined by giving the valid cases, an exception symbol and then the invalid cases. If the invalid cases are permitted to be arbitrary, the metalanguage could define languages which are not context free grammars, including attempts which lead to Russell-like paradoxes, e.g.

```
xx = "AA" - xx ;
```

Is 'AA' an example of xx?

Such licence is undesirable in a standard and therefore the form of the exceptional cases is restricted so that definitions can be proved to be safe.

The exceptional cases must be such that it is possible to define them without using any metaidentifiers. For example the definition of a visible-character on page 9 is valid because space could be replaced by " ".

### Special sequences

Special sequences are allowed to be empty.

A pragmatic use for special sequences is to contain text that informally qualifies the meaning of a syntax rule.

You can shorten the definition of languages containing a large number of lists by defining that:

```
? primary : terminal string ?
```

is equivalent to

```
(primary, {terminal string, primary})
```

APPENDIX B defines special sequences to represent lists and arbitrary permutations.

A note on ambiguous character-strings

Critical readers will have realized that (*) in the metalanguage is ambiguous: it can be read both as (* ) and ( *). Similar problems arise with (:) and (/). These problems are overcome by forbidding any such sequence from appearing in a language definition.

Other syntactic metalanguages

The introduction to this report mentioned that several different metalanguages have previously been used. The syntax of some of them can be defined using the British standard metalanguage.

BACKUS NAUR FORM (Algol 60)

```
BNF syntax = { BNF syntax rule } ;

BNF syntax rule = "<", BNF metaidentifier, ">", "::=",
    BNF definitions list ;

BNF definitions list = BNF single definition,
    { "|", BNF single definition } ;

BNF single definition = { BNF primary }- ;

BNF primary
    = "<", BNF metaidentifier, ">",
    | BNF terminal string | ;

BNF metaidentifier
    = letter, { [ space ], ( letter | digit ) } ;

BNF terminal string = { character }- ;
```

PASCAL (British standard BS 6192)

```
Pascal syntax = { Pascal syntax rule }- ;

Pascal syntax rule = Pascal metaidentifier,
    ( "=", Pascal definitions list
    | Pascal extra defining symbol, Pascal alternative definition
    ), "." ;

Pascal definitions list = Pascal single definition,
    { "|", Pascal single definition } ;

Pascal alternative definition = Pascal definitions list ;

Pascal single definition = { Pascal primary }- ;

Pascal primary = Pascal optional sequence
```

```
    |  Pascal repeated sequence  |  Pascal grouped sequence
    |  Pascal metaidentifier  |  Pascal terminal string  |  ;

Pascal optional sequence  =  '[',  Pascal single definition,  ']'  ;

Pascal repeated sequence  =  '{',  Pascal single definition,  '}'  ;

Pascal grouped sequence  =  '(',  Pascal definitions list,  ')'  ;

Pascal metaidentifier  =  letter,  { [ "-" ],  letter }  ;

Pascal terminal string  =  '"',  { Pascal character },  '"' ;

Pascal extra defining symbol  =  '>' ;
```

A final warning

This beginner's guide is not a complete definition of the syntactic metalanguage; for that you must refer to the British Standard.

# APPENDIX B - SPECIAL SEQUENCES FOR SETS AND LISTS


## SUMMARY

The Standard syntactic metalanguage (BS 6154) includes a facility called 'special sequences' for users who wish to extend the metalanguage for special purposes. This self-contained appendix shows how arbitrary list structures and permutations can be defined concisely using the special sequences.


## INTRODUCTION

The designers of the British Standard syntactic metalanguage could not agree on its extent. Some wanted a small simple metalanguage, others wanted a more comprehensive notation. A compromise was reached where an extension for two-level grammars is suggested in an appendix, and the metalanguage includes "special sequences" whose meaning can be defined by a user.

Two possible extensions are suggested by common questions concerning the metalanguage, "Must I define a list as"

```
item list = item, { separator, item } ;
```

and, "How do I specify a set of items that can occur in any order?"

These notes show how special sequences might be used for these extensions.

The example of Fortran input/output statements given at the end is an illustration of sets in BS6154.

## SYNTAX

```
(* Additional terminal characters *)

  end subset symbol = ">" ;
  intersection operator symbol = "*" ;
  field delimiter symbol = ":" ;
  start subset symbol = "<" ;
  union operator symbol = "+" ;

(* Additional syntax rules for the metalanguage *)

  syntax rule = set definition ;
  special sequence = list | set ;

(* Lists *)

  list = "?", "LIST", field delimiter symbol, list element,
       field delimiter symbol, list separator, "?" ;

    list element = primary - special sequence ;

    list separator = primary - special sequence ;
```

```
(* Sets *)

  set definition = set identifier, defining symbol,
      "?", "SET:?", field delimiter symbol, set expression, "?" ;

  set = "?", "SET", field delimiter symbol,
      set expression, field delimiter symbol, set separator, "?" ;

    set identifier = meta identifier ;

    set expression
        = set term, { union operator symbol, set term } ;

      set term = set primary,
          { intersection operator symbol, set primary } ;

        set primary = set identifier | set element
            | start subset symbol, set expression, end subset symbol
            | start group symbol, set expression, end group symbol ;

          set element = primary - special sequence ;

      set separator = primary - special sequence ;
```

LISTS

An arbitrary list:

```
        ? LIST : list element : list separator ?
```

is equivalent to the syntactic-primary

```
        [ list element, { list separator, list element } ]
```

NOTE - An arbitrary non-empty list can be represented by:

```
        ( ? LIST : list element : list separator ? - empty )
```

where

```
        empty = ;
```

A non-empty list can be defined more concisely as:

```
        ? LIST : list element : list separator ?-
```

Examples

```
    abc = "A" | "B" | "C"
        (*   A   B   C   *) ;
    abc list = "(", ? LIST: abc :?, ")"
        (*  ()  (A)  (Ab)  (AC)  (ABA)  (ABC)  (BBCA)   *) ;
    abc comma list = "(", ? LIST: abc : "," ?-, ")"
        (*  (A)  (A,B)  (A,C)  (A,B,A)  (A,B,C)  (B,B,C,A)  *)
```

## SETS

A set-definition defines the elements of a set and gives a name to the set. The elements of the set are defined by the set-expression, and the name of the set defined by the set-identifier.

A set-expression defines a set that is the union of all the sets defined by the set-terms forming that set-expression.

A set-term defines a set that is the intersection of all the sets defined by the set-primaries forming that set-term.

A set-primary that is a set-identifier defines a set which is the union of all sets defined in a set-definition starting with that set-identifier.

A set-primary that is a set-element defines a set whose sole member is a sequence of symbols represented by the set-element.

A set-element represents any sequence of symbols represented by the primary.

A set-primary that is a start-subset-symbol, followed by a set-expression, followed by an end-subset-symbol defines a subset of the set of symbols defined by the set-expression.

A set-primary that is a start-group-symbol, followed by a set-expression, followed by an end-group-symbol defines the set of symbols defined by the set-expression.

A special-sequence that is a set represents a sequence of symbols. This sequence of symbols is a list of subsequences where
(a) there is one-one relationship between the subsequences and the elements of the set defined by the set-expression, and
(b) every two consecutive subsequences are separated from one another by any sequence of symbols defined by the set-separator.

## Examples

```
(*   Each syntax rule ends with a comment containing examples
     of the symbols or sets defined by that rule   *)

a  =  "A"    (*    A    *)  ;
b  =  "B"    (*    B    *)  ;
c  =  "C"    (*    C    *)  ;


a or b or c  =  "A"  |  "B"  |  "C"
     (*    A    B    C    *)  ;

     (*    Set definitions    *)

a or b or c set definition  =  ? SETDEF: a or b or c  ?
     (*    A    B    C    *) ;
abc set definition  =  ? SETDEF: a + b + c ?
     (*    ABC    ACB    BAC    BCA    CAB    CBA    *) ;
abc subset definition  =  ? SETDEF: <a + b + c>  ?
     (*    Ø    A    AB    AC    CBA    BAC    *)  ;
```

```
        (*    Definitions using sets    *)

    a or b or c set  =  "(",  ? SET: a or b or c set definition :?,  ")"
        (*   (A)  (B)  (C)   *) ;
    abc set  =  "(",  ? SET: abc set definition :?,  ")"
        (*   (ABC)  (ACB)  (BAC)  (BCA)  (CAB)  (CBA)    *) ;
    abc subset  =  "(",  ? SET: abc subset definition  : ?,  ")"
        (*   ()  (A)  (AB)  (AC)  (CBA)  (BAC   *)  ;
    abc comma set  =  "(",  ? SET: abc set definition : "," ?,  ")"
        (*   (A,B,C)  (A,C,B)  (B,A,C)  (B,C,A)
             (C,A,B)  (C,B,A)    *) ;
```

NOTE

It is impossible for special-sequences to be nested or for a separator to be specified directly as a special-sequence-symbol.

This restriction arises because
(1) the special-sequence-symbol both opens and closes a special-sequence, and,
(2) BS 6154 indicates (clauses 4.19 and 4.20) that inside a special-sequence, first-quote-symbols and second-quote-symbols do not necessarily have their normal meaning of delimiting terminal-strings.

## FORTRAN 77 INPUT-OUTPUT STATEMENTS - AN EXAMPLE OF SETS

In several Fortran 77 input-output statements, a set of specifiers may be given by the programmer. The set of specifiers has the following properties:

(a) Each specifier is usually identified by a keyword, followed by equals, followed by an input value or output variable.
(b) The specifier defining the unit-identifier must be specified, but other specifiers are usually optional.
(c) Each specifier is separated from the next by a comma.
(d) When a programmer starts the specification list with the unit-identifier no keyword is necessary for this specifier.

These requirements are satisfied by the following syntax-rules written in BS 6154 with the Set extension defined above.

### Syntax

```
    (*    Nonterminal symbols not defined here:

              array element name,
              character expression,
              integer expr,
              label,
              unit identifier,
              variable name      *)
```

```
    (*   SPECIFIERS   *)

access specifier  =  "ACCESS", "=", character expression ;
blank specifier  =  "BLANK", "=", character expression ;
error specifier  =  "ERR", "=", label ;
file specifier  =  "FILE", "=", character expression ;
form specifier  =  "FORM", "=", character expression ;
iostat specifier
    =  "IOSTAT", "=", (variable name ¦ array element name ) ;
recl specifier  =  "RECL", "=", integer expr ;
status specifier  =  "STATUS", "=", character expression ,
unit specifier  =  "UNIT", "=", unit identifier ;

    (*    SETS OF SPECIFIERS    *)

basic io specifiers
    =  ? SETDEF:  error specifier + iostat specifier  ? ;
close specifiers
    =  ? SETDEF:  basic io specifiers + status specifier  ? ;
open specifiers
    =  ? SETDEF:  basic io specifiers + access specifier
       + blank specifier + file specifier + form specifier
       + recl specifier + status specifier  ? ;

    (*    INPUT - OUTPUT  STATEMENTS    *)

backspace statement  =  "BACKSPACE",  file positioning specification ;

close statement
  =  "CLOSE",
     "(",
        (unit identifier,  [ ",",  ? SET: <close specifiers> : "," ?- ]
        ¦ ? SET: unit specifier + <close specifiers> : "," ?
        ),
     ")"  ;

endfile statement  =  "ENDFILE",  file positioning specification ;

file positioning specification
  =  unit identifier
  ¦  "(",
      (unit identifier,  [ ",",  ? SET: <basic io specifiers> : "," ?- ]
      ¦ ? SET: unit specifier + <basic io specifiers> : "," ?
        ),
     ")"  ;

open statement
  =  "OPEN",
     "(",
        (unit identifier,  [ ",",  ? SET: <open specifiers> : "," ?- ]
        ¦ ? SET: unit specifier + <open specifiers> : "," ?
        ),
     ")"  ;

rewind statement  =  "REWIND",  file positioning specification ;
```

INDEX

CHARACTERS & SYMBOLS

Punctuation characters

, 3

; 2

: see (

! 8

? 5

. 8

Brackets

" 2

' 5

( ) 3
(* *) 5
(/ /) 8
(: :) 8

< > 16

[ ] 4

{ } 4

Delimiters

- 4

* 3
see also (

/ 8
see also (

| 2

= 2